En cada ejercicio se evaluará la eficiencia del código, el uso de identificadores significativos, la indentación, escritura correcta de llaves {}, el uso adecuado de la palabra reservada `const`, el uso de copias, punteros y referencias. Se dispone de tres horas para entregar la prueba y debe realizarse en forma estrictamente individual. Dado que esta prueba evalúa la creación de contededores genéricos, NO se pueden usar contendores de la biblioteca estándar o de cualquier otra biblioteca.

Evaluador de expresiones aritméticas

En un proyecto de software se necesita que la computadora evalúe expresiones aritméticas ingresadas por los usuarios. Se le pide a usted que implemente un eficiente evaluador de expresiones aritméticas. Además es importante que el evaluador sea versátil, porque los usuarios pueden ingresar números de diversa naturaleza, por ejemplo: enteros, reales, fracciones, complejos, polinomios, coordenadas polares, números de precisión arbitraria, y cualquier tipo de datos que reaccione a los operadores aritméticos

Una expresión aritmética es combinación de valores y operadores, que al evaluarse produce un único valor. Los operadores tienen reglas de aridad (cantidad de operandos), asociatividad y precedencia. Para esta versión se necesita que el evaluador sea capaz de trabajar con los cuatro operadores binarios básicos: suma (+), resta (-), multiplicación (*) y división (/). La asociatividad de todos ellos es de izquierda a derecha. La prioridad de esos cuatro operadores es la aprendida en matemática de secundaria. Es decir, la multiplicación y división tienen prioridad sobre la suma y la resta, a menos de que el usuario la cambie usando los operadores paréntesis ().

Indagando por Internet, usted encuentra que se puede implementar un eficiente evaluador de expresiones aritméticas usando dos pilas, una para los operadores y otra para los operandos (valores). El algoritmo que realiza la evaluación de la expresión se conoce en inglés como *shunting yard* y una simplificación es la siguiente:

```
Sea operadores una pila vacía para operadores (caracteres)
Sea valores una pila vacía para valores
Repita:
   Trate de leer un valor de la entrada
   Si pudo leer un valor, apílelo (push) en la pila de valores
   De lo contrario
      Trate de leer un operador de la entrada
      Si no pudo leer un operador, termine el ciclo
      Si el operador es:
         Un paréntesis que abre (:
            Apílelo (push) en la pila de operadores
         Un paréntesis que cierra ):
            Mientras el operador en el tope de la pila no sea el '('
               Llame a aplicar el operador en el tope de la pila
            Retire (pop) el operador '(' de la pila
         Un operador op aritmético +, -, *, /:
            Mientras hayan operadores en la pila con mayor prioridad que op
               Llame a aplicar el operador en el tope de la pila
            Apile (push) a op en la pila de operadores

Mientras hayan operadores pendientes en la pila
   Llame a aplicar el operador en el tope de la pila

El resultado de la expresión estará en el tope de la pila de valores
```

```
Aplicar el operador en el tope de la pila:
   Retire (pop) el tope de la pila de operadores y guárdelo en op
   Retire (pop) el tope de la pila de valores y guárdelo en b
   Retire (pop) el tope de la pila de valores y guárdelo en a
   Retorne el resultado de la operación a op b, de acuerdo al operador op
```

Dado que se quiere que el software en C++ pueda trabajar con muchos tipos de números, usted implementa una plantilla para generar clases evaluadoras de expresiones aritméticas. La plantilla debe ser capaz de hacer funcionar el siguiente programa de pruebas:

```
int main()
{
   std::string dataTypeName;
   std::cin >> dataTypeName;

   if ( dataTypeName == "integer" )
      return std::cout << ExpressionEvaluator<int>().evaluate(std::cin) << std::endl, 0;
   if ( dataTypeName == "real" )
      return std::cout << ExpressionEvaluator<double>().evaluate(std::cin) << std::endl, 0;
   if ( dataTypeName == "fraction" )
      return std::cout << ExpressionEvaluator<Fraction>().evaluate(std::cin) << std::endl, 0;
   if ( dataTypeName == "complex" )
      return std::cout << ExpressionEvaluator<std::complex<double>>().evaluate(std::cin)
         << std::endl, 0;

   return 1;
}
```

Ejemplo de entrada:

```
integer

1 - 2 * ( 3 - 4 ) + 5
```

Ejemplo de salida:

```
8
```

Evaluación

1. [5%] **Plantilla Stack**. Implementa una plantilla de clases `Stack` que puede alojar una cantidad arbitraria de elementos en memoria dinámica discontinua.
2. [5%] **Constructor por defecto** crea una pila vacía. **Método empty()** indica si la pila está vacía.
3. [15%] **Regla de los cinco de `Stack`**. Dado que la clase `Stack` utiliza memoria dinámica, debe evitar fugas de memoria, accesos inválidos y aprovecharse de objetos temporales que no van a ser utilizados más durante la ejecución del programa.
4. [10%] **Método `push()`** inserta un elemento en la pila. **Método `top`** permite acceder al elemento en el tope de la pila. Debe poder permitir modificar el elemento si la pila se puede modificar, o dar acceso de sólo lectura si la pila no se puede modificar.
5. [10%] **Método `pop()` retira y retorna el elemento en el tope de la pila**.
6. [5%] **Plantilla ExpressionEvaluator**. Implementa una plantilla de clases `ExpressionEvaluator` que puede evaluar expresiones aritméticas de cualquier tipo de datos que implemente los operadores aritméticos.
7. [15%] **Regla de los cinco de `ExpressionEvaluator`** ¿Cuál estrategia escogería para aplicar la regla de los cinco a las clases `ExpressionEvaluator`: dejar los métodos por defecto, eliminarlos, o sobrescribirlos? Justifique su decisión e refléjela en el código. En su justificación indique cómo su decisión no genera fugas de memoria y cómo los usuarios de su clase podrían verse beneficiados o limitados.
8. [25%] **Shunting Yard Algorithm**. Implementa la adaptación del algoritmo *shunting yard* que permite leer expresiones del flujo de datos provisto por parámetro, evaluarlas y retornar el valor resultado de la evaluación. Usa su plantilla `Stack` para implementar el algoritmo.
9. [10%] **Métodos utilitarios de `ExpressionEvaluator`**. Implementa métodos utilitarios, como aplicar operadores, y determinar la precedencia de los mismos.